SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 235–251 (1997)

Research

Determinants of Software Maintenance Profiles: An Empirical Investigation

CHRIS F. KEMERER[1] AND SANDRA A. SLAUGHTER[2]*

[1]*Katz Graduate School of Business, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.*
[2]*Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.*

SUMMARY

Software maintenance is a task that is difficult to manage effectively. In part, this is because software managers have very little knowledge about the types of maintenance work that are likely to occur. If managers could forecast changes to software systems, they could more effectively plan, allocate workforce and manage change requests. But, the ability to forecast software modifications depends on whether there are predictable patterns in maintenance work. We posit that there are patterns in maintenance work and that certain characteristics of software modules are associated with these patterns.

We examine modification profiles for 621 software modules in five different business systems of a commercial merchandiser. We find that only a small number of modules in these systems is likely to be modified frequently, and that certain maintenance patterns emerge. Modules frequently *enhanced* are in systems whose functionality is considered strategic. Modules frequently *repaired* have high software complexity, are large in size, and are relatively older. However, modules that have been code generated are less likely to be repaired. Older and larger modules are *restructured* and *upgraded* more frequently. Our results suggest that these characteristics of software modules are associated with predictable maintenance profiles. Such profile information can be used by software managers to predict and plan for maintenance more effectively. In addition, our results suggest the use of code generators as a means of reducing repair maintenance. **© 1997 by John Wiley & Sons, Ltd.**

J. Softw. Maint., **9**, 235–251 (1997)

No. of Figures: 0. No. of Tables: 10. No. of References: 31.

KEY WORDS: software maintenance; software complexity; software management; code generators; strategic systems; cyclomatic complexity

1. INTRODUCTION

While it is well understood that software maintenance requires a significant amount of organizational resources, there exists a relative shortage of quantitative empirical research

* Correspondence: Sandra A. Slaughter, 314A GSIA, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
E-mail: sandras+@andrew.cmu.edu

Contact grant sponsor: Carengie Mellon University
Contact grant sponsor: University of Minnesota

CCC 1040–550X/97/040235–17$17.50
© 1997 by John Wiley & Sons, Ltd.
Received 21 February 1997
Revised 26 April 1997

devoted to improving the performance of those who must manage this activity (Kemerer, 1995). From an organizational perspective, managers find it difficult to plan for maintenance work and to manage maintenance requests because, in part, software managers have very little *a priori* knowledge about the demand for maintenance work and about the ease with which the work can be done.

If managers could forecast the demand for changes to a system, they could more effectively plan releases, allocate workforce and manage change requests. For example, managers could batch changes to modules that are frequently changed to take advantage of scale economies (Banker and Slaughter, 1998). However, signals in the form of past patterns of maintenance are not normally made available to managers due to the effort required to accumulate, report and analyse detailed change request data. Furthermore, the ability to forecast software modifications depends on whether there are predictable patterns in maintenance work.

In this study, we conduct an empirical investigation to determine whether there are predictable maintenance patterns for the business systems of a commercial merchandiser. Our objective is to identify the general factors that are associated with predictable change profiles. This information can then be used by software managers to more effectively plan for maintenance.

2. FACTORS ASSOCIATED WITH SOFTWARE MODIFICATION PATTERNS

2.1. Direct factors

We adopt the IEEE standard definition of software maintenance: *the modification of a software product after delivery to correct faults, improve performance or other attributes, or adapt the product to a modified environment* (IEEE, 1993). Based on this definition, we identify three major classes or profiles of software maintenance work. *Enhancements* include adding, changing or deleting software functionality to adapt to changing business requirements. *Repairs* include corrections to errors in the software code. *Preventive maintenance* includes technical upgrades and restructuring of software code.

There are several patterns that characterize the maintenance of a software system. Over the lifetime of a software system, enhancements to functionality are likely to continue to be done on a fairly regular basis after system installation. Systems that are closest to important areas of the business that change frequently will receive constant enhancement to keep synchronized with the business. A relatively high volume of error correcting activity is likely immediately after system installation as ‘bugs’ are detected when clients use the system in the production environment. After installation, corrective work should stabilize until the system has been in place a number of years (Gefen and Schneberger, 1996). However, as systems reach the end of their useful lives, corrective work may rise due to system entropy (Belady and Lehman, 1976). When systems age, they decay or become disordered due to constant modification that destroys the original structure of the software. Organizations can counter this decay by reorganizing or restructuring their older software (Davis and Olson, 1985, p. 283).

 © 1997 by John Wiley & Sons, Ltd.

At the software *module* level, which in the context of our study is the program level, it is likely that all of the modules in a system will not be modified equally over their lifetimes. Instead, change will be localized to certain modules in the system. This concept of *stress localization* exists because the system is liable to become pathological (or non-operative) if global changes are made rather than limited changes. Rational software designers will therefore design relatively independent modules to minimize the impact of changes and enable the system to remain stable (Parnas, 1972; Warnier 1976, pp. 15–64; Davis, 1987, p. 103). Thus, it is probable that some variant of the 80/20 rule will operate—i.e., 80 per cent of the modifications will be made to 20 per cent of the software modules in a system (Schaefer, 1985).

From a maintenance management perspective, this suggests that certain software modules are much more likely to change over the life of a system. Therefore, identifying the modules and the kinds of modification that they are likely to receive can help software managers in their planning process by reducing uncertainty in the demand for maintenance work.

With all this as a given, the issue then becomes one of how to identify these modification-prone modules. We argue, based on data from the literature and from empirical observations at field sites, that there are certain factors associated with both the amount and type of maintenance work a software module is likely to receive. Three of the factors are:

- functionality,
- development practice, and
- software complexity.

In addition, in our model we *control* for two factors that, although not believed to be direct causes of maintenance, have been suggested to be associated with maintenance effort (Chapin, 1985):

- age, and
- size.

Inclusion of these factors ensures that any relationship discovered between maintenance activity and the first three factors is not due to a simple correlation between the direct factors and these latter two variables. In addition to possibly generating misleading results, these two latter factors may not be under the manager's control.

2.2. Functionality

The kind of functionality implemented in an application is an important driver of change. Applications such as strategic systems that interact more with a volatile and competitive external environment, are likely to be enhanced more frequently to keep synchronization with that environment than are applications such as accounting systems, which are largely internal, relatively stable and narrower in scope (Davis and Olson, 1985, pp. 6–7, 284–285). Thus, we posit:

© 1997 by John Wiley & Sons, Ltd.

Software modules in strategic systems will be *enhanced* **more** frequently than software modules in non-strategic systems.

2.3. Development practice

Another important driver of software change is the particular practice that is employed in the design and development of the software. One of the widely argued benefits of Computer Aided Software Engineering (CASE) tools is that code generators (so-called 'lower CASE' tools) should result in software with higher quality and fewer errors on average than software that has been created by hand. Code generators should thus have a beneficial effect in software maintenance because they reduce the variability in the code and minimize coding errors as the code is automatically created from the software design (Necco, Tsai and Holgeson, 1989; Ehrlich, Lee and Molisani, 1990; Douglass, 1993). Hence, we suggest that:

Software modules that have been code generated will be *repaired* **less** frequently than software modules that have been coded manually.

2.4. Software complexity

A number of studies have linked software complexity with software maintenance performance and software errors (Potier *et al*., 1982; Basili and Perricone, 1984; Banker, Datar and Kemerer, 1991). Software complexity refers to the characteristics of software that make it difficult to understand and to modify (Curtis *et al*., 1979; Basili, 1980). There are multiple dimensions of software complexity (Banker, Datar and Zweig, 1989)—in this study, we focus on one particular dimension of software complexity: cyclomatic or decision density (Gill and Kemerer, 1991; Banker, Davis and Slaughter, 1998).

Decision density refers to the relative amount of decision or control paths in the software per line of code, and has been suggested to be related to maintenance effort. Frequent decision and control flow branching within a software module obscures the relationship between inputs and outputs and increases the cognitive load of the maintenance programmers because they must search among dispersed pieces of code to determine the flow of logic (Ramanujan and Cooper, 1994). Thus, software modules with complex decision branching are liable to contain more errors and will likely be corrected more frequently. Gill and Kemerer (1991) for example, found that this variable is positively associated with maintenance effort, although they cautioned that their results were estimated on a relatively small data sample, and they recommended further empirical testing. Thus, we posit that,

Software modules with high decision complexity will be *repaired* **more** frequently than software modules with low decision complexity.

2.5. Control variable: age

We control for two factors that are associated with change. One of these factors is age, which has been argued to approximate for 'system entropy' in software systems

 © 1997 by John Wiley & Sons, Ltd.

(Vessey and Weber, 1983; Jones, 1989; Gode, Barua and Mukhopadhyay, 1990). For a number of reasons, older software modules will likely receive more enhancements and more repairs. The longer the system is in place, the more the business will change, requiring more enhancements to the system. In addition, as systems age, they tend to become less stable with frequent modification and will thus require repair to remain operational. Because older modules are written using older technologies and techniques, they are prime candidates for restructuring and conversion to newer versions of technology. Thus, we anticipate that:

Older software modules will be *enhanced* **more** frequently than newer software modules.

Older software modules will be *repaired* **more** frequently than newer software modules.

Older software modules will receive **more** *preventive maintenance* than newer software modules.

2.6. Control variable: size

Our other control factor is size. Larger modules will likely receive more enhancements and more repairs than smaller modules, *ceteris paribus*, as larger modules embody greater amounts of functionality subject to change.

Larger modules will likely receive more repairs as well. The larger the module, the more difficult it is to test and validate the module's functionality. This implies that larger modules tend to incorporate more errors. In addition, as business requirements change and new requirements emerge, modules tend to grow in size to incorporate the additional functionality needed. Therefore, larger modules are also good candidates for restructuring to reduce the complexity introduced by size. This leads us to expect that:

Larger software modules will be *enhanced* **more** frequently than smaller software modules.

Larger software modules will be *repaired* **more** frequently than smaller software modules.

Larger software modules will receive **more** *preventive maintenance* than smaller software modules.

By the above arguments, we do not necessarily imply that only very small modules are optimal. This would contradict good design principles, such as, minimize coupling and maximize cohesion. Rather, we suggest that, on average, larger modules will tend to receive more maintenance work of all kinds.

Table 1 summarizes our hypotheses.

3. METHODOLOGY

3.1. Data source

To investigate our hypotheses, we conducted an empirical study of 621 software modules in five different business systems of a commercial merchandiser. These modules were

© 1997 by John Wiley & Sons, Ltd.

Table 1. Factors associated with maintenance profiles

Factor	Measure	Maintenance type	Relationship
Functionality	Strategic systems	Enhancements	+
Development practices	Generated code	Repairs	−
Software complexity	Cyclomatic/decision density	Repairs	+
Age (control variable)	Months	Enhancements, repairs, preventive maintenance	+
			+
			+
Size (control variable)	LOC	Enhancements, repairs, preventive maintenance	+
			+
			+

developed and maintained by the merchandiser's central information systems (IS) department. The IS department was divided into separate development and maintenance groups, with the Development Group working exclusively on new systems and the Maintenance Group supporting and enhancing existing systems. A system at the merchandiser includes a number of software modules that together accomplish a major function, such as accounts receivable processing, payroll or order management.

We collected historical data on the kinds of changes made to these modules from the date they were installed to the date of data collection. The data were extracted from change histories logged by maintenance programmers for each module in the software systems. Programmers recorded the software module creation date and author, the function of the software module, the person making the change, the date of the change, and a description of the change.

3.2. Content coding of module change histories

To identify the number and patterns of changes to the systems, we content analysed the change histories (Krippendorff, 1980). We selected the latent coding technique to identify the underlying meaning in the text. Latent coding is appropriate for this study because we are interested in deducing the kind of maintenance performed based upon the descriptions written by a number of programmers. We developed a coding scheme that classified the change descriptions into one of three maintenance categories (enhancements, repairs and preventive maintenance). Two coders independently coded the change histories after achieving sufficient intercoder reliability for the maintenance categories (agreement between the coders was 100 per cent after several rounds of coding for a sample of modules). After the maintenance categories were coded, the number of changes in each category were summed for each module in a system.

3.3. Measurement

We measured the variables for each module included in our study in the following manner. Enhancements (*SumEnh*) refer to the total number of adds, changes and deletes

 © 1997 by John Wiley & Sons, Ltd.

made to the functionality of the software module over its lifetime. Repairs (*SumRep*) refer to the total number of corrections made to coding errors in the software module over its lifetime. Preventive maintenance (*SumPrev*) refers to the total number of technical conversions and restructurings made to the software module over its lifetime.

The Chief Information Officer and other managers at the merchandiser ranked the systems in terms of their strategic importance from '1' (low importance) to '5' (high importance). Using these rankings, we constructed a variable (*Strg*) for each software module and set its value to the ranking for the system to which the module belonged.

The company's software library system tracked whether a module was the product of a code generator. A binary variable (*CodeGen*) was constructed for each software module to indicate whether it was code generated (1) or written by hand (0).

Cyclomatic or decision density (*DecDen*) was measured using McCabe's cyclomatic complexity metric (McCabe, 1976). This measure determines the number of decision paths in the software module. Each software module was subjected to a commercial software code analysis tool that calculated this metric. To account for the effects of size, McCabe's metric was normalized by dividing it by the number of lines of code for each software module (Gill and Kemerer, 1991; Banker, Davis and Slaughter, 1998).

The age (*AgeMths*) of each software module was calculated as the number of months from the module's initial installation date to the date of data collection.

The size (*SLOC*) of each software module was measured in terms of lines of code. This measure was calculated by the code analysis tool. Each module included in our study was in the COBOL programming language, and therefore many of the normal caveats about using SLOC as a size measure in cross-sectional analyses do not apply here (Banker, Datar and Kemerer, 1991).

3.4. Models

To test our hypotheses, we form three linear regression models:

(1) Enhancements model:

$$SumEnh = \beta_{01} + \beta_{11}\,(Strg) + \beta_{21}\,(CodeGen) + \beta_{31}\,(DecDen) + \beta_{41}\,(AgeMths) + \beta_{51}\,(SLOC) + \epsilon_1 \quad (1)$$

(2) Repairs model:

$$SumRep = \beta_{02} + \beta_{12}\,(Strg) + \beta_{22}\,(CodeGen) + \beta_{32}\,(DecDen) + \beta_{42}\,(AgeMths) + \beta_{52}\,(SLOC) + \epsilon_2 \quad (2)$$

(3) Preventive maintenance model:

$$SumPrev = \beta_{03} + \beta_{13}\,(Strg) + \beta_{23}\,(CodeGen) + \beta_{33}\,(DecDen) + \beta_{43}\,(AgeMths) + \beta_{53}\,(SLOC) + \epsilon_3 \quad (3)$$

© 1997 by John Wiley & Sons, Ltd.

Table 2 summarizes the variables and coefficients included in these models. Note that the β's represent the parameters for the explanatory variables that are statistically estimated in the regression models. The first digit of the subscript for the β parameters represents the designation for the explanatory variable (ranging from 0 to 5). The second digit of the subscript for the β parameters represents the number of the regression model (ranging from 1 to 3). Also, note that β_{01}, β_{02}, and β_{03} represent the intercept terms in each model, and ϵ_1, ϵ_2 and ϵ_3 refer to the disturbance (or error) terms in each model.

We use a multivariate regression to determine the association between maintenance profiles and the explanatory system and module variables for these models. Ordinary least squares (OLS) is used to individually estimate each model. Because each model has identical explanatory variables, OLS is as efficient as generalized least squares for estimation purposes (Greene, 1993, p. 488). Note that these models require local data collection and validation for organizations interested in replicating the results. The precise statistical results reported in the following section apply to the organization examined in this study.

Table 2. Summary of maintenance profile linear regression models

Variable	Coefficient(s)	Definition	How measured
SumEnh	None—dependent variable	Sum of enhancements	Total number of times a module has been enhanced (functionality added, changed or deleted) after installation
SumRep	None—dependent variable	Sum of repairs	Total number of times a module has been repaired after installation
SumPrev	None—dependent variable	Sum of preventive maintenance	Total number of times a module has been restructured or undergone technical conversion after installation
Strg	β_{11}, β_{12}, β_{13}	Strategic importance of functionality	Subjective rating (1–5) of the strategic importance of the system to which the module belongs
CodeGen	β_{21}, β_{22}, β_{23}	Code generated indicator	Binary variable (0–1) indicating whether module is written by hand or code generated
DecDen	β_{31}, β_{32}, β_{33}	Cyclomatic or decision density	McCabe's cyclomatic complexity measure for each module divided by module total lines of code
AgeMths	β_{41}, β_{42}, β_{43}	Age in months	Total number of months from module's installation date to date of data collection
SLOC	β_{51}, β_{52}, β_{53}	Source lines of code	Total number of source lines of code in the module

 © 1997 by John Wiley & Sons, Ltd.

Table 3. Descriptive statistics for the software modules (n = 621)

Variable	Mean per module	Std Dev	Min	Max	Sum	Percentage
SumEnh	7·28	13·38	0·00	137·00	4518·00	83
SumRep	1·09	3·04	0·00	46·00	677·00	12
SumPrev	0·39	1·05	0·00	12·00	245·00	5
SumAll	8·84	16·22	0·00	183·00	5488·00	100

4. ANALYSIS AND RESULTS

4.1. Overall distribution of maintenance work

We began by examining descriptive statistics for all of the software modules. As Table 3 indicates, there were 5 488 total modifications made to the modules, of which 83 per cent were enhancements, 12 per cent were repairs and 5 per cent were preventive maintenance. One module was modified 183 times over a nine year period.

In terms of the percentage of modules modified, 27 per cent of the modules received 80 per cent of the total modifications made; 26 per cent received 80 per cent of the enhancements made; 17 per cent received 80 per cent of the repairs made; and 14 per cent received 80 per cent of the preventive maintenance. This provides strong support for the supposition that a small percentage of software modules receive most of the maintenance work.

We then examined maintenance profiles across each of the business systems. Tables 4 and 5 summarize statistics about each system. Interestingly, of all the possible simple

Table 4. Statistics describing the business systems

System	Accounts receivable	Shipping	Price control	Price management	Mail and phone sales
Year installed	1974	1983	1987	1984	1986
Strategic ranking (1 = low, 5 = high)	1	2	3	4	5
Number of modules	245	48	49	187	92
Percentage of code generated	6·53%	56·25%	34·69%	28·88%	51·09%
Average cyclomatic or decision density (per SLOC)	0·05841	0·04320	0·04372	0·04395	0·05022
Average module age (in months)	87·30	79·5	64·0	71·7	63·5
Average SLOC per module	1267·6	3874·0	2817·7	2672·3	2645·7

© 1997 by John Wiley & Sons, Ltd.

Table 5. Statistics describing the maintenance history

System	Accounts receivable	Shipping	Price control	Price management	Mail and phone sales
Enhancements	1501 total 6·12 per module 6·58 per month	467 total 9·73 per module 3·89 per month	279 total 5·69 per module 3·88 per month	1664 total 8·90 per module 15·40 per month	607 total 6·60 per module 7·23 per month
Repairs	251 total 1·02 per module 1·10 per month	72 total 1·50 per module 0·60 per month	68 total 1·39 per module 0·94 per month	221 total 1·18 per module 2·05 per month	65 total 0·71 per module 0·77 per month
Preventive maintenance	138 total 0·56 per module 0·61 per month	19 total 0·40 per module 0·16 per month	8 total 0·16 per module 0·11 per month	24 total 0·13 per module 0·22 per month	56 total 0·61 per module 0·67 per month

rank ordering of the various rows in both tables (e.g., the rank ordering of 'year installed' is 1, 2, 5, 3, 4 for the five systems) there is only one match, that of 'average cylomatic or decision density' and 'average SLOC per module'. Therefore, the relationships among the variables are not likely to be casually observed without statistical analysis.

Table 6 shows the Pearson correlation matrix for the independent variables in our models. All correlations are significant at the 5 per cent confidence level, with the sole exception of (*SLOC, DecDen*). Note that the correlation between *SLOC* and McCabe's cyclomatic complexity metric is +0·8821 (significant at 5 per cent). In our models, we use a *normalized* measure (*DecDen*) which is McCabe's metric divided by total lines of code. We have chosen this normalized measure so that decision density measures decision complexity, not size complexity. In addition, the measure reduces collinearity problems when *SLOC* is included in the models (Gill and Kemerer, 1991).

Owing to the correlations between variables, tests for multicollinearity were run for the multivariate regression models. For all models, two key indicators of multicollinearity

Table 6. Correlation matrix for independent variables

Variable	*Strg*	*DecDen*	*CodeGen*	*AgeMths*	*SLOC*
Strg	1·0000	—	—	—	—
DecDen	–0·2081*	1·0000	—	—	—
CodeGen	0·3089*	–0·1677*	1·0000	—	—
AgeMths	–0·1866*	0·2361*	–0·2629*	1·0000	—
SLOC	0·2336*	–0·0382	0·8222*	–0·1442*	1·0000

*Indicates significance at 5% confidence level.

 © 1997 by John Wiley & Sons, Ltd.

(variance decomposition proportions and condition indices) are very low. The variance decomposition proportions for each regression coefficient are lower than 0·5, and condition indices for the coefficient matrices are less than nine. Note that Belsley, Kuh and Welsch (1980, pp. 112, 153–159) suggest an upper limit of 0·5 for variance decomposition proportions, and 30 for the condition index. Our figures are well below these limits, indicating that multicollinearity is not a problem in our models.

4.2. Results from the multivariate models

4.2.1. Results: enhancements model

Tables 7 to 9 display the results from our statistical estimation of the three multivariate regression models. We start with the enhancements model, summarized in Table 7.

For the enhancements model all hypotheses are supported, with signs in the predicted directions. Our results indicate that, as anticipated, certain maintenance patterns emerge. Modules in systems whose functionality is considered strategic are significantly more likely to be enhanced. The estimated coefficient for the *Strg* variable in the enhancements model indicates that a one point increase in the strategic rating of the system, holding other explanatory variables constant, leads to a 0·9126 increase in the expected number of enhancements for a software module. In addition, a strong relationship between enhancement activity and both *DecDen* and *CodeGen*, although not explicitly hypothesized, is also supported by the data. A potential *ex post facto* explanation for the positive relationship between *DecDen* and enhancements is that when modules are enhanced frequently, there is more opportunity to introduce complexity. Or, parts of the system that are 'decision rich' may be subject to greater numbers of external changes. The adjusted R^2 for the model is 0·38, and can be interpreted that the model explains about 38 per cent of the variance in the enhancement activity at this site.

Table 7. Regression results: enhancements model. $SumEnh = \beta_{01} + \beta_{11}\ (Strg) + \beta_{21}\ (CodeGen) + \beta_{31}\ (DecDen) + \beta_{41}\ (AgeMths) + \beta_{51}\ (SLOC) + \epsilon_1$

Variables	Coefficient	Predicted sign	Estimate	*t*-value	*p*-value (one-tailed)
Intercept	b_{01}		–9·8531	–6·349	0·0001
Strg	b_{11}	+	0·9126	3·174	0·0008
DecDen	b_{21}		56·0795	2·798	0·0026
CodeGen	b_{31}		–18·7022	–10·348	0·0001
AgeMths	b_{41}	+	0·0794	8·381	0·0001
SLOC	b_{51}	+	0·0048	15·235	0·0001
R^2			0·3852		
Adjusted R^2			0·3802		
F-test (model)				77·0706 (*F*-value)	0·0001

© 1997 by John Wiley & Sons, Ltd.

4.2.2. Results: repairs model

For the repairs model all hypotheses are supported, with signs in the predicted directions. The overall model has a similar strength of fit as the enhancements model. We find that modules frequently repaired have high software complexity, are large in size, and are relatively older. The estimated coefficient for the *DecDen* variable in the repairs model indicates that an increase of one decision statement per line of code leads to an increase of 11·7581 in the expected number of repairs for a software module, holding other explanatory variables constant.

However, modules that have been code generated are significantly less likely to be repaired. The estimated coefficient for the *CodeGen* variable in the repairs model indicates that the expected number of repairs is 4.3961 lower for code generated modules. One possible interpretation of this result is that programmers prefer not to update generated code, so they take another approach or live with the problem. Another potential interpretation of this result is that the code generator is effective for development but provides poor mechanisms for maintenance activities, and therefore, programmers minimize maintenance of code generated modules. However, at our particular field research site, the Maintenance Group was obligated to meet service level requirements for each system. Each year at this organization, end users for a particular system set service level (uptime) requirements in excess of 99 per cent. This means that the Maintenance Group is responsible for making whatever corrections are necessary to the system in order that it meets these service level guidelines. Thus, programmers could not refuse or avoid maintenance to code generated modules or there would be a negative impact on service levels.

There is another interesting characteristic of the code generator used at this field research site. The code generator is a back-end CASE tool that generates COBOL code. It has the ability to incorporate 'custom' hand-written code in the generated code. For example, in a three day benchmarking test of CASE tools, the code generator was found to require heavy custom COBOL coding outside of the tool to implement the functional

Table 8. Regression results: repairs model. $SumRep = \beta_{02} + \beta_{12}\ (Strg) + \beta_{22}\ (CodeGen) + \beta_{32}\ (DecDen) + \beta_{42}\ (AgeMths) + \beta_{52}\ (SLOC) + \epsilon_2$

Variables	Coefficient	Predicted sign	Estimate	*t*-value	*p*-value (one-tailed)
Intercept	b_{02}		−1·5047	−4·041	0·0001
Strg	b_{12}		0·0289	0·419	0·3376
DecDen	b_{22}	+	11·7581	2·446	0·0074
CodeGen	b_{32}	−	−4·3961	−10·140	0·0001
AgeMths	b_{42}	+	0·0077	3·392	0·0003
SLOC	b_{52}	+	0·0011	14·838	0·0001
R^2			0·3138		
Adjusted R^2			0·3082		
F-test (model)				56·2439 (*F*-value)	0·0001

© 1997 by John Wiley & Sons, Ltd.

requirements posed (Computerworld, 1991). However, the rigid format of the tool was found to assist in code readability. At our research site, a significant amount of custom coding was needed to meet the requirements of the end users. The custom code was implemented in the generated code using called subroutines. Managers and programmers indicated that most of the maintenance was to this custom code, and was made to the called subroutines which were outside of the tool and resided in common libraries. The tool provides little support for updates to custom code.

While user service level agreements dictated changes to systems, these changes impacted primarily the custom code which was outside of the code generating tool. Programmers changed the custom code in a non-CASE environment, and then checked it against the generated code to ensure compatibility. Because of the difficulties inherent with modifications in a CASE environment, programmers tended to make enhancements in batches which had the effect of reducing the counts of enhancements to code-generated modules. This is consistent with the negative coefficient for CodeGen in our Enhancements Model. However, for repair work, programmers were required to make whatever repairs were necessary to ensure compliance with uptime agreements. Thus, they could not batch repairs in groups but had to correct errors as they occurred. This provides support for our result that code-generated modules have fewer errors on average than non-code-generated modules, *ceteris paribus*.

4.2.3. Results: preventive maintenance model

All of the hypothesized relationships among the variables in the preventive maintenance model are found to be significant. We find that older and larger modules tend to be restructured and upgraded more frequently on average. However, the overall fit of this model is not as strong as the enhancement model or repair model, with an adjusted R^2 of only 0·18. In addition to the hypothesized relationships, a somewhat weaker relationship between *Strg* and preventive maintenance activity is also supported by the data.

Table 9. Regression results: preventive maintenance model. $SumPrev = \beta_{03} + \beta_{13}\ (Strg) + \beta_{23}\ (CodeGen) + \beta_{33}\ (DecDen) + \beta_{43}\ (AgeMths) + \beta_{53}\ (SLOC) + \epsilon_3$

Variables	Coefficient	Predicted sign	Estimate (n = 621)	t-value	p-value (one-tailed)
Intercept	b_{03}		–0·3514	–2·508	0·0001
Strg	b_{13}		–0·5435	–2·094	0·0184
DecDen	b_{23}		–0·0848	–0·047	0·4813
CodeGen	b_{33}		0·1818	1·114	0·1328
AgeMths	b_{43}	+	0·0088	10·351	0·0001
SLOC	b_{53}	+	0·0001	2·746	0·0031
R^2			0·1915		
Adjusted R^2			0·1850		
F-test (model)				29·1375 (F-value)	0·0001

© 1997 by John Wiley & Sons, Ltd.

Thus, the results from all three models demonstrate that certain characteristics of software modules can be associated with predictable maintenance patterns.

4.3. '80/20' results

In examining the raw data, we find that there is rough support for the widely cited '80/20' rule that about 80 per cent of all work is caused by only about 20 per cent of all modules (Schaefer, 1985). For the five systems in this data set, 80 percent of the modifications were made to between 20 and 30 per cent of the modules. To test this relationship further, we classified the software modules into two groups: high/low complexity and high/low maintenance. To identify the high complexity group, we ordered the modules from high to low on the *DecDen* measure, and then assigned a '1' to the top 20 per cent of modules and a '0' to the remaining 80 per cent of the modules. To identify the high maintenance group, we ordered the modules from high to low based on *SumAll* (the sum of enhancements, repairs and preventive maintenance). We assigned a '1' to the top 20 per cent of modules and a '0' to the remaining 80 per cent of the modules.

We then cross-tabulated the modules using the high complexity and high maintenance groups and computed a chi-square to test whether there is a significant relationship between high complexity and high maintenance. Table 10 shows the results from the cross-tabulation.

The chi-square value of 34·056 is significant at $p = 0{\cdot}00001$ and indicates that there is a significant relationship between the high maintenance and high complexity categories. The number of modules classified as both high maintenance and high complexity is almost twice that expected.

We conducted the 80/20 analysis for our other explanatory variables, including high age versus high maintenance, high code generation versus high maintenance, high size versus high maintenance and high strategic importance versus high maintenance. We found a significant and positive relationship between high age and high maintenance (chi-square value of 61·537, $p = 0{\cdot}00001$). The number of modules classified as both high maintenance and high age is more than twice that expected. There is a significant negative relationship between high code generation and high maintenance (chi-square value of 7·743, $p =$

Table 10. '80–20' cross-tabulation of modules

Modules grouped by amount of maintenance		Modules grouped by complexity		
		Low actual (expected)	High actual (expected)	Total number (%)
Low	actual	421	76	497
	(expected)	(398)	(99)	(80%)
High	actual	76	48	124
	(expected)	(99)	(25)	(20%)
Total	number	497	124	621
	(%)	(80%)	(20%)	(100%)

 © 1997 by John Wiley & Sons, Ltd.

0·00539). The number of modules classified as both high maintenance and high code generation is only about half that expected. For high size versus high maintenance, we found a non-significant relationship (chi-square value of 1·429, $p = 0·23197$). Interestingly, this implies that there is no relationship between size and maintenance volume for the top 20 per cent of modules in both groups. Finally, we also found no significant relationship between high maintenance and high strategic importance (chi-square value of 0·480, $p = 0·448826$) for the top 20 per cent of modules in both groups.

5. CONCLUSIONS

The results of the statistical analysis suggest that maintenance activity does follow predictable patterns and therefore can be the subject of more rigorous managerial planning than may be widely believed. The three variables, *Strg*, *DecDen* and *CodeGen* are all factors that can be measured *a priori*, and therefore serve as useful planning tools. In addition, it should be noted that the hypotheses regarding these variables were supported by the models after having controlled for both size and age, two commonly believed factors in maintenance activity. Therefore, the results found here cannot be seen as an artefact due to possible correlation with these other factors.

Going further, these results suggest possible strategies for not simply *planning* for software maintenance activity, but actively attempting to *reduce* it. The *DecDen* and *CodeGen* variables are the result of activity that takes place during software development. Our models indicate that, on average, modules with high levels of decision density are significantly associated with both frequent enhancement and frequent repair. This suggests that organizations may wish to implement guidelines for upper bounds of *DecDen* during development and could recommend that software not exceed these guidelines before it is placed into production. Similarly, during maintenance releases the code can be re-examined to ensure that the maintenance work has not inadvertently increased this measure of static complexity.

Organizations could also use these results to argue for the beneficial effects on software quality of so-called lower CASE tools. Our analysis supports the result that, at our research site, code-generated modules were repaired significantly less on average than non-code-generated modules. This result cannot be explained by the notion that maintenance programmers avoided repairing these modules, because the programmers were bound to fix errors as they occurred in order to meet user service level requirements.

Our results from the 80/20 analysis suggest that organizations do not need to invest significant amounts of resources in implementing sophisticated techniques to identify maintenance-prone modules. A simple ordering of modules from high to low using the *DecDen* or *AgeMths* measures of complexity and age, respectively, can be used to classify the modules into the group (20 per cent) which is likely to receive the majority of maintenance work. For our data set, this group includes 124 of the 621 modules. These modules can then be selected for special treatment, which might include assignment to a more senior maintainer, or even to rewriting. The costs of such a procedure would need to be traded-off against the potential benefits to determine whether either restructuring or reassignment is more cost effective. We emphasize that local data collection and validation

© 1997 by John Wiley & Sons, Ltd.

is needed to calibrate the maintenance profile models for organizations that wish to replicate our results.

Overall, the measure of any such study such as this one is to argue for a more rigorous and quantitative approach to software maintenance management, an activity of considerable economic significance to modern organizations.

Acknowledgements

This study was funded in part by a Faculty Development Grant from Carnegie Mellon University and by a Grant from the Quality Leadership Center at the University of Minnesota. The authors thank Wee Lin Sim for her data coding efforts. We gratefully acknowledge the helpful comments of the anonymous reviewers.

References

Banker, R., Datar, S., and Kemerer, C. (1991) 'A model to evaluate variables impacting productivity on software maintenance projects', *Management Science*, **37**(1), 1–18.

Banker, R., Datar, S. and Zweig, D. (1989) 'Software complexity and maintainability', in *Proceedings International Conference on Information Systems*, ACM, New York, NY, pp. 247–255.

Banker, R., Davis, G. and Slaughter, S. A. (1998) 'Software development practices, software complexity, and software maintenance effort: a field study', *Management Science*, forthcoming.

Banker, R. and Slaughter, S. A. (1998) 'A field study of scale economies in software maintenance', *Management Science*, forthcoming.

Basili, V. (1980) 'Quantitative software complexity models: a panel summary', in *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 232–233.

Basili, V. and Perricone, B. (1984) 'Software errors and complexity: an empirical investigation', *Communications of the ACM*, **27**(1), 42–52.

Belady, L. and Lehman, M. (1976) 'A model of large program development', *IBM Systems Journal*, **15**(1), 225–252.

Belsley, D., Kuh, E. and Welsch, R. (1980) *Regression Diagnostics*, John Wiley and Sons, New York, NY, 292 pp.

Chapin, N. (1985) 'Software maintenance: a different view', in *AFIPS Conference Proceedings*, Volume 54 NCC, AFIPS Press, Reston, VA, pp. 328-331.

Computerworld (1991) 'Pansophic short on CASE; development and maintenance report card', *Computerworld*, **25**(37), 37.

Curtis, B., Sheppard, S., Milliman, P., Borst, M. and Love, T. (1979) 'Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics', *Transactions on Software Engineering*, **SE-5**(2), 96–104.

Davis, W. (1987) *Systems Analysis and Design: A Structured Approach*, Addison-Wesley, Reading MA, 415 pp.

Davis, G. and Olson, M. (1985) *Management Information Systems: Conceptual Foundations, Structure and Development*, McGraw-Hill, New York, NY, 693 pp.

Douglass, D. (1993) 'The costs and benefits of CASE', *I/S Analyzer*, **31**(6), 1–16.

Ehrlich, W., Lee, S. and Molisani, R. (1990) 'Applying reliability measurement: a case study', *IEEE Software*, **7**(2), 56–64.

Gefen, D. and Schneberger, S. (1996) 'The non-homogeneous maintenance periods: a case study of software modifications', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 134–141.

Gill, G. and Kemerer, C. (1991) 'Cyclomatic complexity density and software maintenance productivity', *Transactions on Software Engineering*, **17**(12), 1284–1288.

Gode, D., Barua, A. and Mukhopadhyay, T. (1990) 'On the economics of the software replacement

 © 1997 by John Wiley & Sons, Ltd.

problem', in *Proceedings of the Eleventh International Conference on Information Systems*, University of Copenhagen, Denmark, pp. 159–170.
Greene, W. (1993) *Econometric Analysis*, 2nd ed., MacMillan Publishing Company, New York, NY, 791 pp.
IEEE (1993) *IEEE Standard for Software Maintenance*, IEEE, New York, NY, 39 pp.
Jones, C. (1989) 'Software enhancement modelling', *Journal of Software Maintenance*, **1**(1), 91–100.
Kemerer, C. (1995) 'Software complexity and software maintenance: a survey of empirical research', *Annals of Software Engineering*, **1**(1), 1–22.
Krippendorff, K. (1980) *Content Analysis: An Introduction to its Methodology*, Sage Publications, Newbury Park, CA, 191 pp.
McCabe, T. (1976) 'A complexity measure', *Transactions on Software Engineering*, **SE-2**(4), 308–320.
Necco, C., Tsai, R. and Holgeson, K. (1989) 'Current usage of CASE software', *Journal of Systems Management*, **40**(5), 6–11.
Parnas, D. (1972) 'On the criteria to be used in decomposing systems into modules', *Communications of the ACM*, **15**(12), 1053–1058.
Potier, D., Albin, J., Ferreol, R. and Bilodeau, A. (1982) 'Experiments with computer software complexity and reliability', in *Proceedings Sixth International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 94–103.
Ramanujan, S. and Cooper, R. (1994) 'A human information processing approach to software maintenance', *Omega*, **22**(2), 185–203.
Schaefer, H. (1985) 'Metrics for optimal maintenance management', in *Proceedings Conference on Software Maintenance—1985*, IEEE Computer Society Press, Los Alamitos, CA, pp. 114–119.
Vessey, I. and Weber, R. (1983) 'Some factors affecting program repair maintenance: an empirical study', *Communications of the ACM*, **26**(2), 128–134.
Warnier, J. (1976) *Logical Construction of Programs*, Van Nostrand-Reinhold, New York, NY, 230 pp.

Authors' biographies:

Chris F. Kemerer holds the David Roderick Chair in Information Systems at the University of Pittsburgh. He received his doctorate from Carnegie Mellon University, and was an Associate Professor at MIT's Sloan School of Management prior to accepting his current position. Dr. Kemerer's research interests include management and measurement issues in information systems and software engineering. He has published articles on these topics in leading professional and academic journals. He is a former Principal of American Management Systems, Inc., the Arlington, Virginia-based software development and consulting firm.

Sandra A. Slaughter is an Assistant Professor at Carnegie Mellon University in the Graduate School of Industrial Administration. She received her doctorate in information systems from the University of Minnesota. Her research focuses on productivity and quality improvement in information systems development and maintenance. Her dissertation, *Software Development Practices and Software Maintenance Performance*, received the ICIS Best Dissertation Award in 1995. Dr. Slaughter has worked for Hewlett-Packard as an information systems planner and for Allen-Bradley Company and Square D Corporation as an information systems project manager.

© 1997 by John Wiley & Sons, Ltd.